INRIA

INSTITUT NATIONAL DE RECHERCHJE EN INFORMATIQUE ET AUTOMATIQUE

MYOAN: an Implementation of the KOAN Shared Virtual Memory on the Intel Paragon

Gilbert Cabillic, Thierry Priol, Isabelle Puaut

N° 2258

Avril 1994

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

ISSN 0249-6399

MYOAN: an Implementation of the KOAN Shared Virtual Memory on the Intel Paragon

Gilbert Cabillic, Thierry Priol, Isabelle Puaut

Programme 1- Architectures parallèles, bases de données, réseaux et systèmes distribués

Projet LSP

Rapport de recherche n°2258- Avril 1994

24 pages

Abstract: KOAN is a shared virtual memory (SVM) system initially designed and implemented for the Intel iPSC/2 supercomputer. The main features of KOAN are its support for both strong and relaxed consistency. In addition, KOAN offers three mechanisms increasing the performance of applications using SVM: page broadcasting, page locking and a mechanism which eliminates double faults. This document describes the design choices, implementation and first performance measures of an implementation of KOAN, named MYOAN, on the Intel Paragon XP/S.

Key-words: Shared virtual memory, Paragon XP/S, KOAN.

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : (33) 99 84 71 00 - Télécopie : (33) 99 38 38 32

MYOAN : une Réalisation de la Mémoire Virtuelle Partagée KOAN sur Intel Paragon

Résumé : KOAN est une mémoire virtuelle partagée initialement conçue pour la machine parallèle iPSC/2 d'Intel. Une caractéristique de KOAN est de supporter à la fois un protocole de maintien de cohérence forte et une définition relâchée de cohérence. En outre, KOAN propose trois mécanismes permettant d'améliorer l'efficacité des programmes: la diffusion de pages, le verrouillage de pages, et la résolution du problème de la double faute. Ce document relate une réalisation de la mémoire virtuelle partagée KOAN, nommée MYOAN, sur la machine parallèle Intel Paragon XP/S. Les choix de conception de cette mémoire virtuelle partagée, ainsi qu'une description de sa réalisation et de premières mesures de performance sont présentés.

Mots-clé : Mémoire virtuelle partagée, Paragon XP/S, KOAN.

1 Introduction

Shared Virtual Memory (SVM) [Li & Hudak 89] is a software abstraction of shared memory on an architecture without hardware support for shared memory, such as, for example, a network of workstations or a distributed memory multicomputer. This abstraction is attractive because it simplifies programming; processors can access both local and remote data using the standard *read* and *write* operations. In most SVM systems, consistency of shared data is managed by using distributed versions of multicache consistency protocols (e.g., [Li & Hudak 89], [Michel 90], [Puaut *et al.* 91]) that scale the unit of sharing to a virtual memory page in order to increase performance. Another option for improving performance is to define less restrictive definitions of memory consistency than the standard (strong consistency) criterion, stating that "*reads return the most recent write"* [Censier & Feautrier 78]. Propositions for relaxed definitions of memory consistency include *weak consistency* [Dubois *et al.* 86], *release consistency* [Gharachorloo *et al.* 90], *causal consistency* [Ahamad *et al.* 91], and *entry consistency* [Bershad & Zekauskas 91].

KOAN [Lahjomri & Priol 92], [Lahjomri & Priol 91] is a SVM facility initially designed for an Intel iPSC/2 Hypercube. An interesting feature of KOAN is its support for multiple consistency protocols: both strong consistency and a relaxed form of consistency permitting multiple writers on the same page are supported. Moreover, a shared virtual memory region can change its consistency semantics dynamically. This paper describes the design choices, implementation and first performance measurements of MYOAN, which is an implementation of KOAN for the Intel Paragon supercomputer. This paper is organized as follows. Section 2 gives an overview of the Intel Paragon supercomputer hardware and software. Section 3 then describes the main features of KOAN. The implementation of MYOAN on the Paragon is detailed in Section 4. Early performance measurements of the implementation are given in Section 5. Finally, conclusions are outlined in Section 6.

2 Overview of the Intel Paragon Supercomputer

2.1 System Hardware

The Intel paragon supercomputer [Int 93b] is a distributed memory multicomputer composed of a large number of processors, called *nodes*, linked by a high speed node interconnect network. Each node is a separate computer with two i860 processors [Int 89] and at least 16M bytes of memory. On each node, one processor is aimed at running applications while the other is a dedicated communication processor. Some nodes (*service* nodes) are dedicated by the system administrator to interactive applications, while other nodes (*compute* nodes) run compute-intensive applications. The nodes are interconnected by a 200Mb/sec network and are arranged in a two-dimensional array. Each node interfaces to this network through special hardware that monitors the network and extracts only those messages addressed to its attached node. Messages addressed to other nodes are passed without interrupting the node's computations. Some

nodes are equipped with a SCSI interface, Ethernet interface, or other I/O connections. The operating system running on each node makes transparent to nodes without I/O hardware, the access to I/O devices though the use of the node interconnect network.

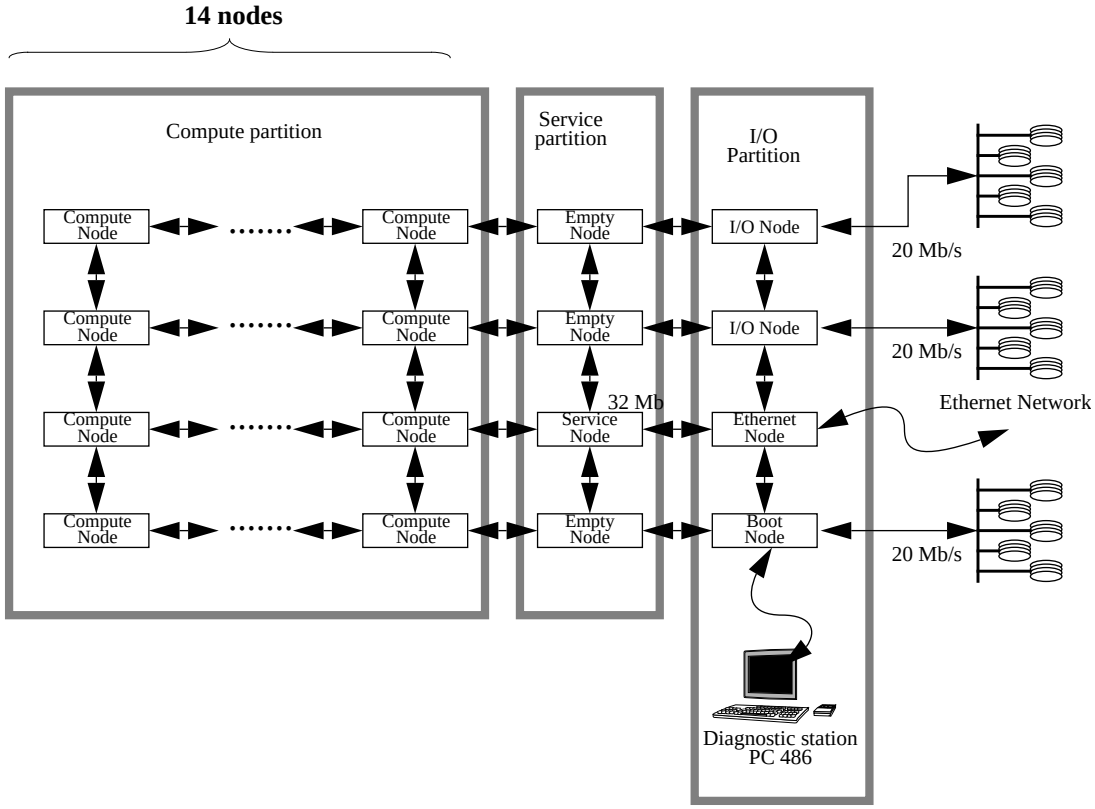

Fig. 1: The Intel Paragon hardware configuration at IRISA

The hardware configuration we are equipped with at IRISA (see Fig. 1) is made of 56 compute nodes, 3 service nodes, 3 I/O nodes and 3 RAID disks of 4.8 Gbytes each.

2.2 System Software

The nodes run the Paragon OSF/1 operating system*, based on the Mach OSF/1 operating system from the Open Software Foundation [Loepere 93b]. OSF/1 is a version of the Unix operating system; Paragon OSF/1 is an extended version of OSF/1 [Loepere 93a] with enhancements to support parallel processing.

* Precisely, we use OSF/1 release 1.0.4, which is based on a the Mach 3.0 kernel NORMA_MK13.26 R1.1.4 and the Unix server 1.1 R1.1.4.

Like standard OSF/1, Paragon OSF/1 [Zajcew *et al.* 93] is made of the Mach 3.0 micro-kernel and a server implementing Unix features. The micro-kernel provides the following entities: *tasks*, *threads*, *ports* and *messages*. *Threads* are lightweight execution entities. A task includes an address space, as well as a set of ports that are shared by the threads running inside the task. Finally, threads communicate by sending *messages* on communication *ports*. In the following we simply call *process* a task and its associated threads. Memory management relies on paged virtual memory. However, unlike traditional virtual memory designs, the kernel does not implement all the virtual memory software: user-mode tasks, called *external pagers*, have the ability to participate in the implementation of virtual memory management.

Paragon OSF/1 provides all the standard features of OSF/1 with extensions to provide a single image system across multiple nodes: all the nodes share a single file system and have equal access to the system's I/O devices. The single system image does not combine all the nodes' memory into a single address space. Rather, each process has its own (virtual) address space; memory pages that do not fit in physical memory are paged to disk. As in most multi-user systems, the address spaces of different processes are independent, unless the processes make special shared virtual memory calls to explicitly share part of their memory.

3 KOAN : a Shared Virtual Memory Facility for the Intel iPSC

KOAN [Lahjomri & Priol 91], [Lahjomri & Priol 92], [Lahjomri 93] is a shared virtual memory facility running on the Intel iPSC/2 Hypercube [Intel 88]. It differs from the work described in [Li & Schaefer 89] in that it runs in kernel mode within the NX/2 kernel [Pierce 88] which provides basic memory and process management facilities. KOAN allows *regions* of virtual memory to be shared by processes running on distinct nodes of the hypercube. An interesting feature of KOAN is its ability to support multiple consistency protocols; both strong (atomic) consistency and a relaxed form of consistency, that permits multiple writers on the same page, are supported. Moreover, the consistency semantics of a region may change dynamically. The reader is referred to [Lahjomri 93] for a detailed description of the implementation and performance of KOAN; only a brief description is given hereafter.

3.1 Support for Strong Consistency

The default memory consistency semantics for memory regions in KOAN is strong consistency. Each read at a given address returns the last value written at the same address. Strong consistency is maintained by implementing an invalidation-based protocol. A given page may be replicated in the nodes' physical memories only if it is protected against writes (it is in read-only access mode), while pages in read-write access mode cannot be replicated. Should a process attempt to write in a read-only page, its replicas are invalidated before the write can proceed.

When a page fault or an access violation on a page occurs, the current access mode of the page as well as the nodes having a copy of the page in their physical memories, have to be

identified. The knowledge of the current status of pages of a shared region may either be *centralized* on a node, or *distributed* among a set of nodes. In order to avoid the bottleneck of a centralized approach, KOAN distributes the information on a set of nodes: for a given region, each node registers the current status of a subset of the region's pages. In order to implement the consistency protocol easily, a *fixed* distributed scheme was selected; given the identification of a virtual page, the node having information on that page can be known statically.

3.2 Support for Relaxed Consistency: Multiple-writers Protocol and Page merging

Many relaxed consistency models have been proposed in order to increase the performance of SVM systems. KOAN proposes a relaxed form of consistency for parallel processes satisfying the well-known Bernstein conditions [Bernstein 66], which are used for verifying if processes can execute in parallel. Let I_i (*Input set*, or *Read set*) be the set of variables needed for executing process P_i and O_i (*Output set*, or *Write set*) be the set of variables modified by P_i, the Bernstein conditions (see left part of Fig. 2) state that two processes P_1 and P_2 can execute in parallel if they act on independent data.

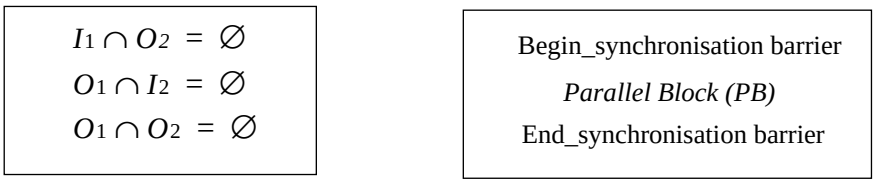

Fig. 2: Bernstein conditions and parallel blocks

If we call *parallel block* the sequences of statements satisfying the Bernstein conditions, two synchronizations barriers are required: one before the beginning of the parallel block, and the other after the end of its execution. As, due to Bernstein conditions, there is no data dependence between the elements of a parallel block, the new values of the variables modified by each element need not be seen immediately by the others. This permits KOAN to increase the performance of applications as detailed below.

The consistency semantics of the memory region associated to a parallel block is set to *multiple-writers* at the beginning of the block. As shown in Fig. 3, during the execution of a parallel block, each page of such a region is replicated in read-write access mode in the physical memories of the processors that write into the page. At the end of a parallel block, the consistency semantics of the region is reset to *strong consistency*. Consequently, each page of the

region must have again a single read-write copy; this copy is obtained by merging the replicas created during the parallel block's execution.

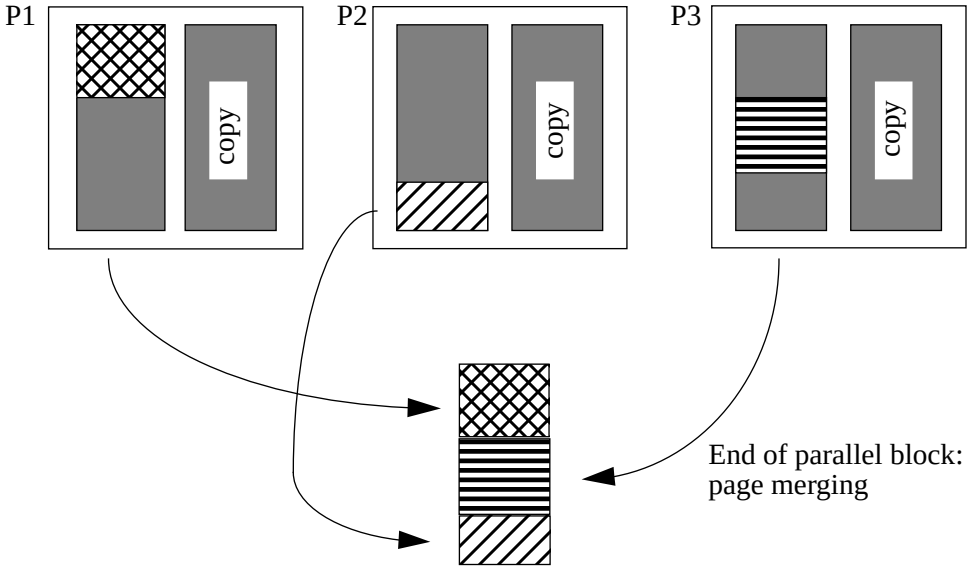

Fig. 3: Implementation of multiple-writers consistency semantics

Merging can be done in two ways. One method consists in finding the bytes that were modified during the parallel block by applying an *exclusive or* between each replica of the page and a base copy of the page made before the beginning of the parallel block. Another method, that is used in the current implementation of KOAN, consists in having each writer process modify a distinct range of addresses within the shared region, which eases page merging.

Multiple-writers consistency semantics eliminates conflicts when parallel processes modify independent data belonging to the same virtual page, and thus removes the resulting performance overhead due to page faults and invalidations. Performance gains obtained by the KOAN multiple-writers consistency semantics are detailed in [Lahjomri 93]

3.3 Other Features

In addition to the support of a relaxed type of consistency for data-independent parallel blocks, KOAN provides three mechanisms (page broadcasting, page locking and a mechanism for eliminating the double faults) for improving the performance of applications using SVM. These mechanisms are sketched in the following paragraphs. KOAN also offers synchronisation tools by means of *events* and *mutual exclusion semaphores*; for space consideration, synchronization tools are not described in this document; more details can be found in [Lahjomri 93].

Page broadcasting

The *page broadcasting* mechanism was introduced for increasing the performance of parallel programs that exhibit a *producer-consumer* scheme, where one producer process writes data in a region while several consumer processes, after synchronizing with the producer, read the data. Using strong consistency by means of an invalidation-based consistency protocol (see Fig. 4) results in page faults for consumer processes, as well as message exchanges required for transferring the page(s) from the producer to the consumers.

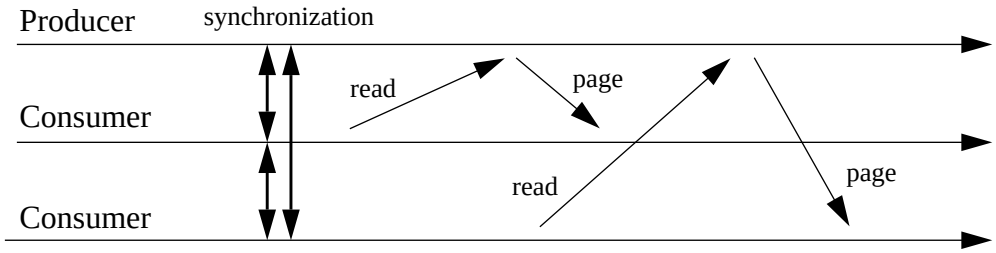

Fig. 4: Inefficiency of an invalidation-based protocol on a producer-consumer scheme

The basic idea of the page broadcasting mechanism of KOAN is to broadcast pages that have been modified by the producer to the consumers at the end of the production phase. This can be implemented efficiently if the underlying network provides a broadcast or multicast facility. The page broadcasting mechanism is provided through two routines: *begin_broadcast* and *end_broadcast*. A call to *begin_broadcast* initiates a production phase, and makes the producer record the list of pages it will modify; a call to *end_broadcast* initiates the beginning of a consumption phase and causes the pages that were modified by the producer to be sent to the consumers.

Page locking

The *page locking* mechanism allows to handle efficiently processes running on distinct nodes that modify concurrently data belonging to the same virtual page, and hence to implement efficiently mutual exclusion on data structures belonging to a single virtual page. Locking a page consists in wiring the page in the physical memory of the node that calls the page

locking routine; a *locked* page had a single read-write copy that cannot be moved until it gets unlocked.

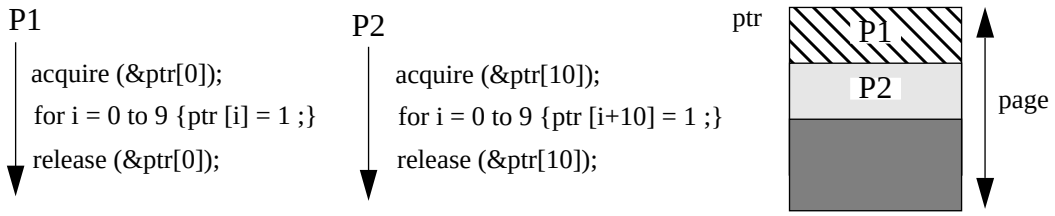

Fig. 5: Use of the page locking mechanism

Fig. 5 shows how to use KOAN's page locking mechanism. A process wires a page in memory by calling the routine *acquire(addr)*, where addr is a virtual address within the page to be locked. The page is unlocked by calling the routine *release(addr)*. On the example, *ptr* begins at a page boundary and the whole array of integers fits in a single virtual page; hence, as an optimization, a single call to *acquire* is issued. However, when no special care is taken for having a data structure on a single page, a call to *acquire* must be issued for each word of the data structure.

Elimination of double faults

A *double fault* is a read page fault occurring on a node and immediately followed by a write page fault on the same page and node. Double faults occur in many numerical applications when statements of the form *x := f(x,y)* are executed.

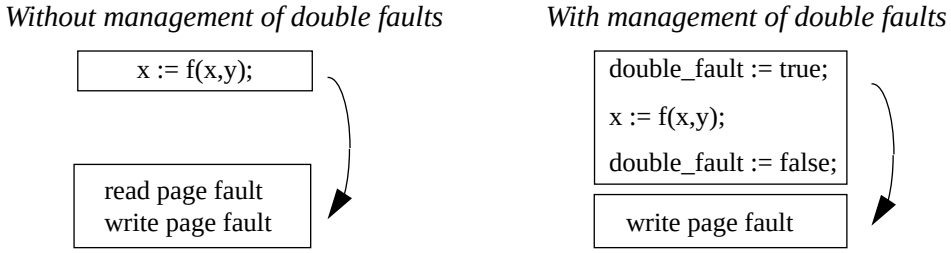

Fig. 6: The issue of double faults and its solution in KOAN

KOAN eliminates the performance penalty resulting from double faults by adding a boolean variable, say *b*, shared between KOAN and user processes. When a read page fault occurs, if *b* is true, the page fault is transformed into a write page fault. This mechanism is illustrated in Fig. 6.

4 MYOAN: an Implementation of KOAN on the Paragon

This section describes MYOAN, which is an implementation of KOAN on the Intel Paragon XP/S. The description is divided into three steps. First, memory management in the Paragon OSF/1 operating system is described. Then, the design choices for building MYOAN on top of this operating system are highlighted. Finally, the implementation of MYOAN is sketched.

4.1 Memory Management in Paragon OSF/1

The Mach 3.0 kernel [Loepere 93b] provides mechanisms to support large, potentially sparse virtual address spaces. Each task has an associated address map (maintained by the kernel) which controls the translation of virtual addresses into physical addresses. The contents of the entire address space of a given task is most likely not completely resident in physical memory.

Unlike traditional virtual memory systems, the mechanisms that exist for using physical memory as a cache for the virtual address space of tasks are not entirely implemented in the Mach kernel. Any given region of virtual memory is backed on a *memory object*. A *memory manager* task, also called *external pager,* provides the policy governing the relationship between the image of a set of pages while cached in memory (the physical memory contents of a memory region) and the image of that set of pages when not so cached (the abstract *memory object*). The Mach kernel comes with a default memory manager providing basic non-persistent memory objects that are zero-filled initially and paged against system paging space.

The Mach kernel and a memory manager communicate through message passing, as shown in Fig. 7. The figure shows the different states of a memory object, as well as messages exchanged between the kernel and the external pager that make the state of a memory object change. When a user needs to map a memory object on a region of virtual addresses, he/she calls the *vm_map* kernel routine, with the port identifying the memory object as a parameter. This port is owned by the external pager managing the memory object, and will be used by the kernel for notifying the external pager of events occurring on the memory object (e.g., page faults). When the user calls the *vm_map* routine, the kernel sends an initialization message, named *memory_object_init* to the external pager. If a memory access is attempted on a missing page subsequently to a *vm_map*, the kernel sends a message *memory_object_data_request* to the memory manager. The memory manager replies to the kernel by sending a message

memory_object_data_unavailable if the page has not been created yet. A message *memory_object_data_supply* embedding a copy of the page is sent otherwise.

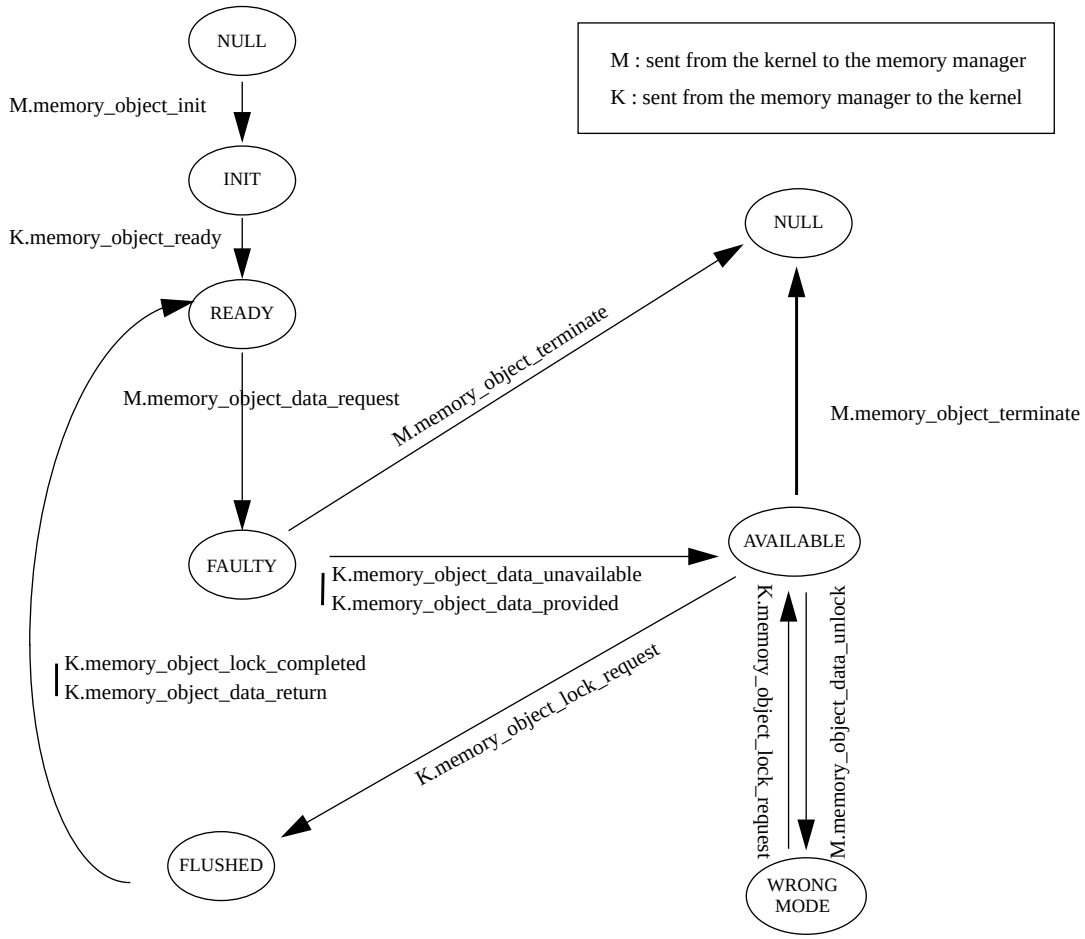

Fig. 7: States of a memory object

When the memory manager needs to invalidate a page for maintaining consistency, it sends a message *memory_object_lock_request* to the kernel. The kernel replies by sending a message *memory_object_data_return* containing the page if the page was modified, and a message *memory_object_lock_completed* indicating that the invalidation is completed. When the kernel needs to change the access mode of a page, it sends a message *memory_object_data_unlock* to the manager. Finally, when a memory object is unmapped, the kernel sends a message *memory_object_terminate* to the memory manager for destroying the data structures that have been allocated for the memory object.

4.2 Design Choices

Building a SVM facility on top of Paragon OSF/1 raises two issues. It must be chosen whether to implement the SVM software at the kernel or at the user-level. In addition, one of the two ways that exist on Paragon OSF/1 for inter-node communication has to be selected. These two issues are discussed in the following paragraphs.

4.2.1. Kernel-level versus user-level implementation

There are two approaches for implementing a SVM facility in Paragon OSF/1. The first one consists in building the SVM software at the user-level by building an external pager. The alternative solution is to modify the Mach kernel for adding a SVM facility. Although the KOAN SVM was implemented at the kernel-level for performance reasons, MYOAN is implemented as an external pager for the following reasons. First, we are convinced that an operating system kernel should stay small and should include only facilities required for a wide range of applications. Second, implementing a SVM by modifying an existing kernel, although better from the standpoint of efficiency, has a severe portability disadvantage. Should the internal structure of the kernel change, the SVM code has also to be changed. Finally, implementing a SVM by extending an existing kernel requires to have the source code of the kernel, which turns out to be difficult in many situations. A SVM facility implemented at the user-level is less efficient than a similar facility implemented at the kernel-level. However, some micro-kernels such as Chorus [Rozier *et al.* 88] offer the ability to execute servers (*actors* in Chorus) at the kernel level, and then removes a part of the performance penalty while keeping advantages of user-level implementation.

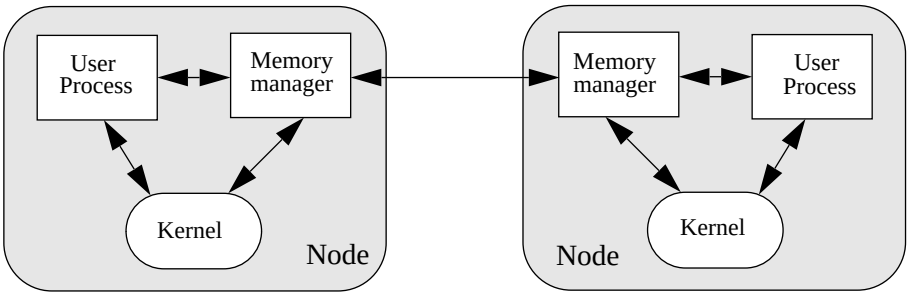

Fig. 8: Logical structure of MYOAN

The logical structure of MYOAN is shown in Fig. 8. There is one memory manager per node and per application using SVM. On the figure, arrows depict communications. On page faults and protection violations, the user process communicates with the kernel; the kernel then communicates with the local memory manager for retrieving the page; these communications use the standard Mach/OSF interprocess communication facility. In addition, on a node, user processes communicate with the local memory manager in order to call MYOAN routines. Fi-

nally, memory managers communicate with each others in order to maintain consistency of shared regions. The choice of a communication strategy for making memory managers communicate with each others, as well as for making the application communicate with its memory manager is crucial as it greatly influences the performance of the SVM facility; this issue is discussed below.

4.2.2. Interprocess communication: Norma IPC versus NX

There are two ways by which threads can communicate with each others on the Paragon OSF/1 system. One way consists in using the standard Mach/OSF interprocess communication facility, called NORMA [Langerman 93], and noted *Norma IPC* in the following, which runs in kernel mode and can be used both for intra-node and inter-node communications. An alternative way to make threads communicate with each others is to use the Intel NX message passing library [Int 93a], simply called *NX* in the following, that provides functions for sending or receiving (synchronously or asynchronously) typed messages between nodes, as well as forking processes. Note that at the time this report is written, the second i860 processor of each node is not exploited yet as a dedicated communication processor by the Paragon OSF/1 operating system. Consequently, interprocess communication overhead is likely to decrease in future releases of the operating system for both NX and Norma IPC. This paragraph compares the performance of Norma IPC and NX for inter-node and intra-node communications in order to choose the most appropriate tool for communication between memory managers, and between application processes and memory managers.

Performance is measured thanks to one procedure, called MinArg. MinArg takes two integer arguments and returns an integer result, which is the minimum of the two parameters. This procedure is called by sending a message containing the parameters (using either Norma IPC or NX) to the process implementing the procedure, and then waiting for a message embedding the results. Measurements were made in multi-user mode. When using NX, the synchronous message passing primitives *csend* and *crecv* were called.

Inter-node communication

Performance measurements of inter-node communication using NX and Norma IPC for MinArg are given in Fig. 9. Measurements were obtained by implementing loops with up to

10000 remote calls to MinArg. The given values are obtained by dividing the total elapsed time of the loop by the number of calls to MinArg.

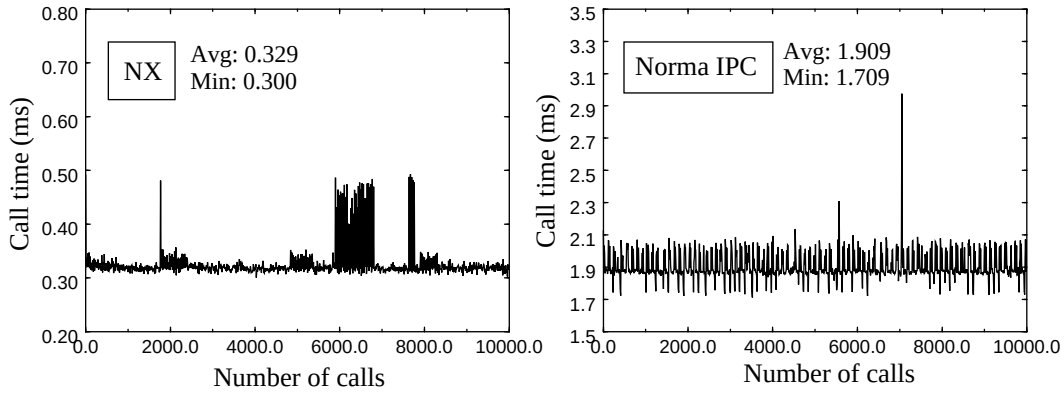

Fig. 9: Performance of inter-node communications using NX and Norma IPC

The average time for calling MinArg is 0.329 ms using NX, while 1.909 ms are required when using Norma IPC. As Norma IPC is about 6 times slower than NX on inter-node communications, our shared virtual memory facility uses NX when inter-node communications are required (e.g., when two memory managers communicate for processing a page fault).

Performance of intra-node communication

On a given node, communicating threads can share the same address space (i.e. they belong to the same task) or not. Fig. 10 shows the call time of MinArg when interacting threads belong to different tasks.

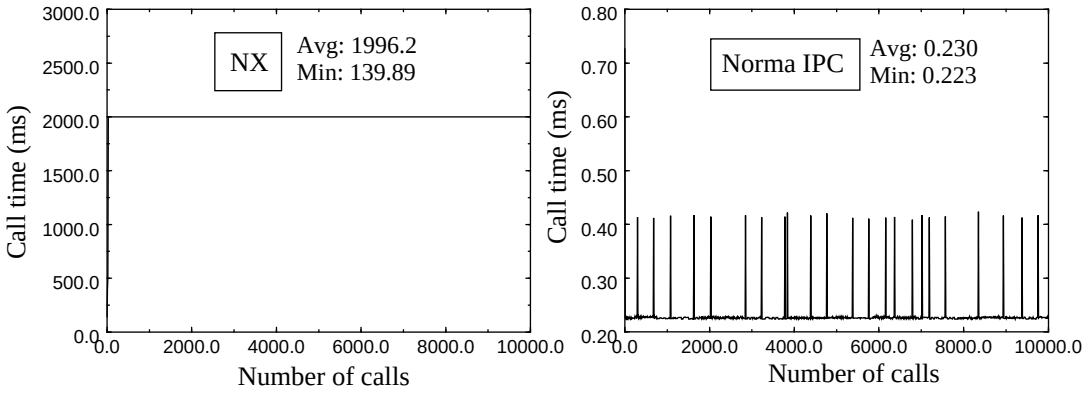

Fig. 10: Performance of inter-process (intra-node) communications using NX and Norma IPC

These measures show that NX must not be used for inter-process communication when the two communicating processes run on the same node; 2 seconds are required for each call. This average call time of two seconds when using NX is disastrous. While at the time this report is written we cannot fully explain the reason of such an unexpected behavior, we suspect the NX library to make intensive use of busy waiting for testing the arrival of messages. On the other hand, when using Mach IPC, 230 µs are required in average for inter-process communication.

When communicating threads belong to the same task (see Fig. 11) NX exhibits the same disastrous performance (about two seconds per call) as for inter-process communication on the same node. When using Norma IPC, a minimum of 53 µs is required for MinArg, which is about four times less than if the interacting threads belong to distinct tasks.

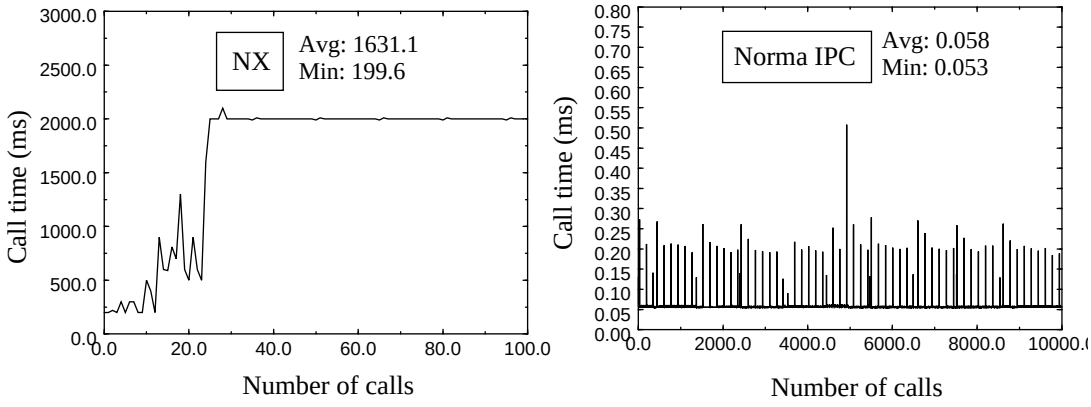

Fig. 11: Performance of intra-process (intra-node) communications using NX and Norma IPC

These results make us avoid using NX when interacting threads run on the same node. In addition, as Mach IPC is implemented efficiently when communicating threads share the same address space, we chose to have one memory manager per application and per node, and to implement on a given node the application's process and the memory manager inside the same Mach task. This permits, as detailed later, to solve the double fault issue and to implement synchronization tools efficiently.

4.3 Implementation

4.3.1. Overview of the implementation

MYOAN is implemented as a set of memory managers. For a given application, there is one memory manager per node on which the application runs, and the memory manager shares the application's address space. A MYOAN memory manager communicates with the kernel

it is running on by using Mach IPC, while memory managers communicate with each others through NX. Each memory manager has the structure shown in Fig. 12.

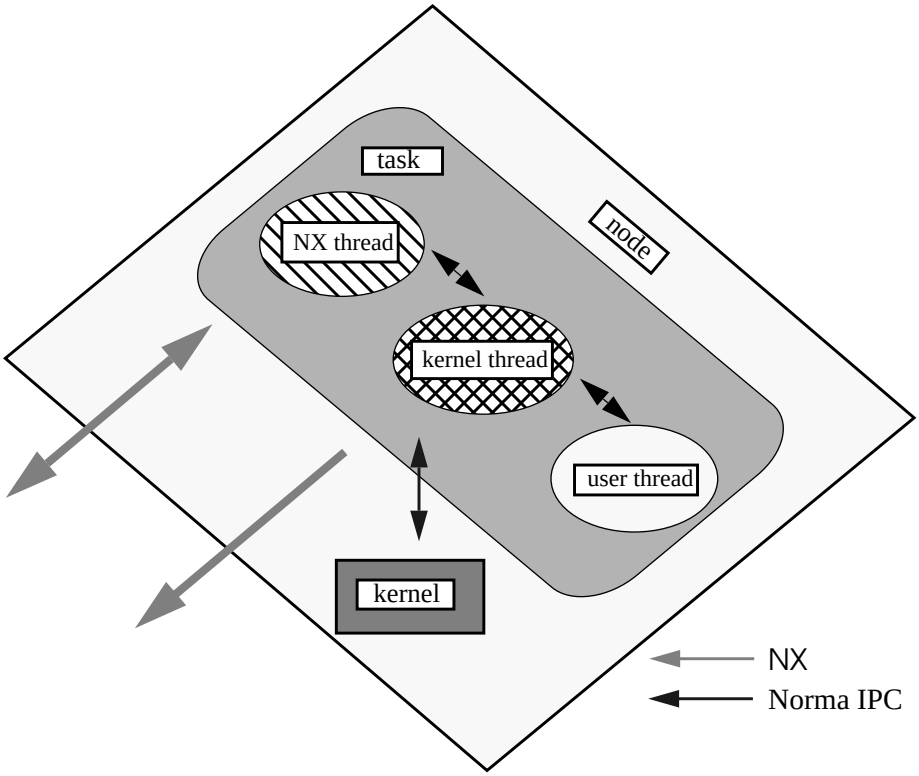

Fig. 12: Structure of a memory manager

A memory manager is made up of two Mach threads. The first thread receives messages from the kernel via Mach IPC, while the second receives requests from remote pagers via NX. Both threads also sends back pages to the kernel after page faults processing. A Mach port is allocated to each shared region when it is created. A Mach *port_set* allows to listen for all the page fault messages coming from the kernel.

4.3.2. Implementation of strong consistency

A fixed distributed scheme, similar to the scheme described in [Li & Hudak 89] is used for managing strong consistency. Given the identification of a page, a statically known external pager, called the *manager* of the page, maintains the list of nodes having a copy of the page. The function which is applied for determining the manager of a page is selected when a shared region is mapped into a process address space; the function is either *Modulo* (page *p* is managed by the memory manager of node *p mod n*, where *n* is the number of nodes used by the application) or *Block* (page *p* is managed by the memory manager of node *p div n*). The node

having a read-write copy of the page, or the first node having a read-only copy of a page is called its *owner*.

When a page fault occurs on a node, the kernel sends a message to the local external pager. This external pager then notifies the page's *manager* of the occurrence of a fault. The manager either invalidates the copies of the page if the page has read-only replicas, or gets the page from its *owner* node if it is in read-write mode. An example of interactions between nodes on a write page fault is shown in Fig. 13, where initially the page has a single read-write copy on a node that is different from the page's manager.

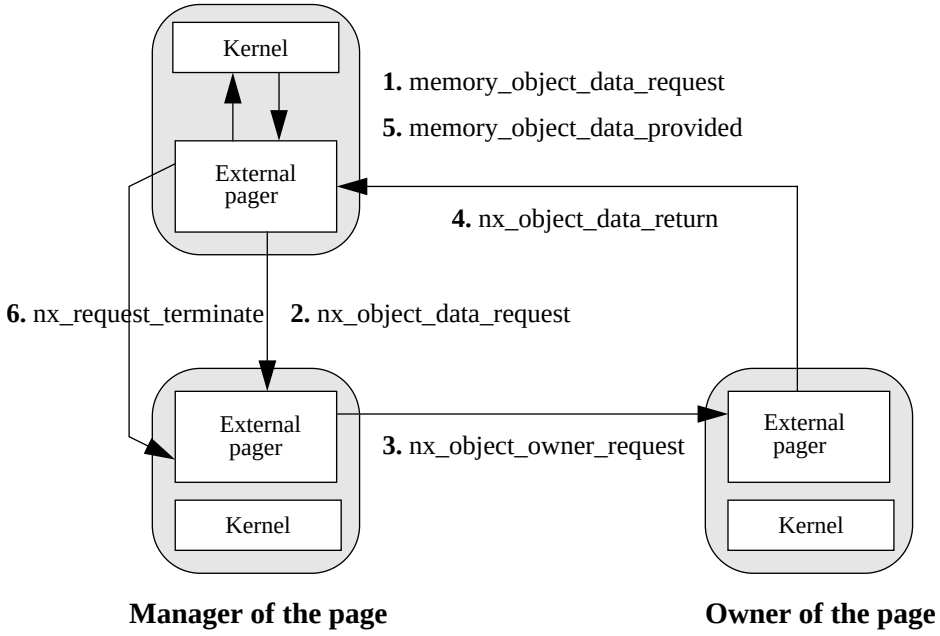

Fig. 13: An example of interaction between external pagers when solving a page fault

On the figure, a write page fault occurs on a node N. The kernel of N notifies the memory manager of the page fault via Mach IPC (step 1 in the figure). The external pager then asks the page to the page's manager (step 2). This is done by using NX for performance reasons. As in our example the page is stored in read-write mode on a node different from the manager's node, the manager sends a message to the page's owner (step 3). The page's owner then sends the page to the faulting node and invalidates its copy (step 4). When the faulting node receives the page, it sends the page to the kernel (step 5), and then notifies the manager of the termination of the page fault processing (step 6). On this example, three NX inter-node communications, as well as two inter-process communications on the same node are required.

The manager of a page registers a *page descriptor.* The descriptor identifies the mode (read-only or read-write) of the page and the list of nodes having copies of the page in their physical memory. A flag embedded in the page descriptor indicates if a fault is currently being handled on the page (e.g., the manager has sent invalidation messages and is waiting for messages acknowledging the invalidations). When a message indicating a page fault is received while this flag is set, the manager links the page fault request in a list which is embedded in the page descriptor. Upon receipt of a message *nx_request_terminate* indicating the end of the treatment of a page fault, the first request of the request list (if any) is handled.

4.3.3. Implementation of the multiple-writers protocol

At the beginning of a parallel block, each component of the parallel block calls a routine called *begin_ppsd* with the identifier of the shared region that will be modified as a parameter. This routine implements the synchronization barrier required before the execution of the parallel block. In addition, the routine multicasts a message to the managers of the pages belonging to the shared region. Upon receipt of these messages, the managers make a copy of the pages that will be modified within the parallel block.

During the parallel block execution, each time a page fault occurs, the manager of the page returns the base copy of the page done before the beginning of the parallel block. The identifier of the nodes having a (read-write) copy of the page are remembered in the page descriptor by the page's manager.

At the completion of the parallel block, each component of the parallel block calls the routine *end_ppsd*, with the identifier of the region that has been modified as a parameter. The routine implements the end synchronisation barrier, but also sends a message to the managers of the pages belonging to the modified region. Upon receipt of this message and for a given page, the manager of the page asks its owners to return their copy of the page and to invalidate it. The received pages are then merged; the bytes that were modified are identified by doing an *exclusive or* with the base copy of the page done before the beginning of the parallel block.

4.3.4. Implementation of the other features of KOAN

The implementation of page broadcasting, page locking, and the mechanism for avoiding double faults are briefly discussed in the three following paragraphs.

Page broadcasting

At the beginning of a production phase, each process involved in the producer-consumer scheme calls the *begin_broadcast* routine with the following parameters: the identifier of the region that is shared between the producer and the consumers, the range of virtual addresses to be modified by the producer process, and the identifier of the producer's running node. When called on the producer's node, *begin_broadcast* makes the memory manager of the node register the list of pages that will be modified by the producer. When *begin_broadcast* is called on one of the consumers' nodes, a message embedding the identifier of the node is sent to the

producer's node. This way, the memory manager of that node registers the list of future consumers.

At the end of the production phase, each process involved in the producer-consumer scheme calls the *end_broadcast* routine with the identifier of the shared region as a parameter. The execution of *end_broadcast* on the producer's node makes the memory manager of that node multicast a message containing the modified pages to the memory managers of the consumers' nodes. Upon receipt of such a message, a consumer's memory manager stores the page in the physical memory of the node.

Page locking

Page locking consists in wiring a page in the physical memory of the node that calls the page locking routine, called *acquire*. The execution of this routine on a node sets to true a flag in its local page descriptor indicating that the page is locked. When a page fault occurs on the page, the memory manager of the node requests a read-write copy of the page from its owner, which invalidates the copies of the page if it is replicated in read-only mode. When a message requesting a page is received by a memory manager and the page is locked, the message is kept in the page descriptor until the page is unlocked.

When a page is unlocked subsequently to a call to *release*, the memory manager of the node on which the routine is called checks that no message requesting the page is kept in the page descriptor. If so, the flag indicating whether the page is locked is reset. On the other hand, if a node requested the page, the page is sent to that node before resetting the flag.

Elimination of double faults

Double faults are avoided through the provision of a routine called *myoan_double_fault* which takes as parameter a virtual address identifying the page on which a double fault is expected. The execution of this routine sets a flag, embedded in the page descriptor of the node on which the routine is called. When a page fault occurs, the value of the flag is tested. If the flag denotes true, every page fault is treated as a write page fault and consequently, a message notifying a write page fault is sent to the page's manager. If the flag is false, a message notifying a page fault with the actual required access mode is sent to the manager. Another routine is provided for resetting the flag that is used for notifying the pagers of double faults.

4.3.5. Implementation status

The implementation status of MYOAN is the following. The basic features of KOAN (strong consistency and multiple writers protocol), as well as the mechanism eliminating double faults are implemented. Page broadcasting and page locking are currently under implementation. Furthermore, in the current implementation, page-outs are not fully treated; whenever a kernel sends a page-out message to a memory manager before invalidating the page in main memory, the memory manager keeps a copy of the page in a dynamically allocated area whose swap is managed by the default memory manager. A full treatment of page-outs will require to write pages on disk or to move them in the physical memory of nodes having free memory.

5 Performance

The basic operation costs of MYOAN are shown in Table 1. Measurements were obtained by making loops of 200 page faults on pages belonging to the same shared region and managed by the same external pager. Page size is 8 Kb and experiments were done in multi-user mode.

Operation	Cost (ms)
1. Write page fault (page creation on manager's node)	0.957
2. Write page fault (page creation on node ≠ manager)	1.656
3. Write page fault (remote RW page, owner = manager)	4.068
4. Write page fault (remote RW page, owner ≠ manager)	4.098
5. Read page fault (remote RO page, owner ≠ manager)	4.478
6. Access violation (RO page, 32 replicas)	12.656

Table 1: Basic operation costs of MYOAN

The first number was obtained by attempting to write a value in a not yet created page that is managed by the memory manager of the node on which the fault occurred. The second case is identical to the first one, except that the manager of the page runs on a different node. The third number was obtained by writing into a page having a single copy on the page manager's node. The fourth case is identical to the third one, except that the page's owner and the page's manager run on distinct nodes. The fifth number was obtained by attempting to read a page with a read-only copy owned by a node different from the manager. Finally, the last number was obtained by attempting to write in a read-only page when 32 replicas of the page exist.

Handling a page fault takes from 0.957 to 12.656 ms depending on the current status of the page. This first performance measurements were obtained in a first (not yet optimized) implementation of MYOAN. We are convinced that a careful optimization of our code will improve the performance significantly.

6 Conclusion

In this document we described the design issues, implementation and first performance measurements of MYOAN, which is an implementation of the KOAN shared virtual memory on the Paragon XP/S supercomputer. Unlike KOAN, and due to the structure of the operating

system running on the Paragon, based on the Mach micro-kernel, we chose to run the SVM software at the user-level by building an external pager. Although this choice introduces performance overhead compared to kernel-implemented SVM facilities, this leads to higher portability. Performance of communication software plays a crucial role in the efficiency of a SVM facility. For that purpose we chose to use the NX library instead of the standard Mach IPC (which is about 6 times slower than NX on inter-node communications) for implementing communications between memory managers. First performance measures of MYOAN are promising; with a not yet optimized code, handling a page fault takes from 1 to 12 ms. We are currently finishing and optimizing the implementation of MYOAN and are evaluating its performance. Performance measurements will be compared with those obtained with the KOAN implementation made on the iPSC/2 computer.

Acknowledgments

The work described in this paper is supported by Intel SSD under an External Research and Development Program (INRIA contract no. 193C21431318012). We wish to thank Valérie Issarny for her helpful comments on earlier drafts of this paper.

References

[Ahamad *et al.* 91] M. Ahamad, P.W. Hutto & R. John. « Implementing and programming causal distributed shared memory ». *Proc. of 11th International Conference on Distributed Computing Systems*, pages 274–281, May 1991.

[Bernstein 66] A.J. Bernstein. « Analysis of programs for parallel processing ». *IEEE Transactions on Computers*, pages 746–757, October 1966.

[Bershad & Zekauskas 91] B.N. Bershad & M.J. Zekauskas. « Midway : Shared memory parallel programming with entry consistency for distributed memory multiprocessors ». Research Report CMU-CS-91-170, *Department of Computer Science, Carnegie-Mellon University, Pittsburgh*, September 1991.

[Censier & Feautrier 78] L.M. Censier & P. Feautrier. « A new solution to coherence problems in multicache systems ». *IEEE Transactions on Computers*, 27(12):1112–1118, December 1978.

[Dubois *et al.* 86] M. Dubois, C. Scheurich & F. Briggs. « Memory access buffering in multiprocessors ». *Proc. of 13th Annual International Symposium on Computer Architecture*, pages 434–442, Tokyo, June 1986.

[Gharachorloo *et al.* 90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A.Gupta & J. Hennessy. « Memory consistency and event ordering in scalable shared memory multiprocessors ». *Proc. of 17th Annual International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, May 1990.

[Int 89] Intel Corporation. *i860 64-bit Microprocessor Programmer's Reference Manual*, 1989.

[Int 93a] Intel Corporation. *Paragon OSF/1 C system calls reference manual*, 1993.

[Int 93b] Intel Corporation. *Paragon User's Guide*, 1993.

[Intel 88] Intel. « The intel iPSC/2 system ». *Third Conference on Hypercube Concurrent Computers and Applications*, pages 843–846, Pasadena, California, 1988.

[Lahjomri & Priol 91] Z. Lahjomri & T. Priol. « KOAN : a shared virtual memory for the iPCS/2 hypercube ». Research report, *INRIA*, September 1991.

[Lahjomri & Priol 92] Z. Lahjomri & T. Priol. « KOAN : a shared virtual memory for the iPCS/2 hypercube ». *CONPAR/VAPP92*, September 1992.

[Lahjomri 93] Z. Lahjomri. *Conception et réalisation d'un mécanisme de mémoire virtuelle partagée sur un machine multiprocesseur à mémoire distribuée*. Thèse de doctorat, université de Rennes I, 1993.

[Langerman 93] A. Langerman. *Norma IPC version two : requirements*. Open Software Foundation and Carnegie Mellon University, 1993.

[Li & Hudak 89] K. Li & P. Hudak. « Memory coherence in shared virtual memory systems ». *ACM Transactions on Computer Systems*, 7(4):321–357, November 1989.

[Li & Schaefer 89] K. Li & R. Schaefer. « A hypercube shared virtual memory ». *Proc. of 1989 International Conference on Parallel Processing*, pages 125–131, 1989.

[Loepere 93a] K. Loepere. *OSF Mach Kernel Interface*. Open Software Foundation and Carnegie Mellon University, 1993.

[Loepere 93b] K. Loepere. *OSF Mach Kernel Principles*. Open Software Foundation and Carnegie Mellon University, 1993.

[Michel 90] B. Michel. « Gothic memory management : a multiprocessor shared single level store ». Research Report 1202, *INRIA*, March 1990.

[Pierce 88] P. Pierce. « The nx/2 operating system ». *Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390, Pasadena, California, 1988.[

[Puaut *et al.* 91] I. Puaut, M. Banâtre & J.P. Routeau. « Early experiences with the Gothic distributed operating system ». *Proc. of the Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS II)*, pages 271–282, Atlanta, Georgia, March 1991.

[Rozier *et al.* 88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, P. Léonard, S. Langlois & W. Neuhauser. « The Chorus distributed operating system ». *Computing Systems*, 1(4):305–370, 1988.

[Zajcew *et al.* 93] R. Zajcew, P. Roy, D. Black, C. Peak, P. Guedes, B. Kemp, J. LoVerso, M Leibensperger, M. Barnett, F. Rabii & D. Netterwala. « An osf/1 unix for massively parallel multicomputers ». *Proceedings of the 1993's Usenix Winter Conference*, pages 37–56, 1993.